Points-clés sur les vecteurs en C++

Michel Simatic et Amina Guermouche

Télécom SudParis

7 avril 2025

Plan

1. Avant-propos : Notion de référence en C++
2. Vecteurs
3. Erase, tri et lambda expressions

Avant-propos : Notion de référence en C++

1. → Avant-propos : Notion de référence en C++
2. Vecteurs
3. Erase, tri et lambda expressions

Paramètres d'une fonction (cf. Chapitre 8)

- Passage par valeur :

```
void print(vector<double> v) {
  for(auto i : v)
      cout << i << ',';
}
```

- NB :
  - Copie des paramètres
  - Modifications locales à la fonction
  - **MAIS**
    - Et si la variable est un vecteur de 8 Mo ?
    - En plus, on ne fait qu'afficher ce vecteur…

Passage par const-reference

```
void print(const vector<double>& v) {
  for(auto i : v)
    cout << i << ',';
}

void f() {
  vector<double> v1(100);
  ...
  print(v1);
}
```

- `v` fait référence (*refers back*) à la variable définie ailleurs.
- Pas de copie.
- Étant donné que `v` n'est pas modifiée dans cette fonction, le mot clé `const` permet d'éviter qu'on la modifie par erreur.

Passage par référence

```
void print(vector<double>& v) {
  for(int i=0; i < v.size(); i++)
    ++ v[i];
}
```

- `v` fait référence (*refers back*) à la variable définie ailleurs.
- Pas de copie.
- Une modification du paramètre est visible à l'extérieur de la fonction.

Quel passage de paramètre choisir ?

- Objets très petits (un ou deux doubles ou entiers) : **par valeur**
- Objets larges pas modifiés : **par const-reference**
- Préférez retourner un résultat au lieu de passer le paramètre par référence.
- Utilisez le passage par référence seulement quand il le faut. Par exemple :
  - Pour manipuler les vecteurs et autres objets larges,
  - Pour des fonctions modifiant plusieurs objets car il n'est pas possible de retourner plus d'un objet. Il est cependant déconseillé d'écrire des fonctions modifiant plusieurs objets (Exemple à ne pas suivre ??? TP "Six qui prend" !)

Et le passage par adresse dans tout ça ?

- Préférez le passage par référence.
- Utilisez le passage par adresse uniquement quand c'est nécessaire

```
f(T *t) // A utiliser si t peut être nullptr, car une référence ne peut pas être nulle.
```

Quelques remarques sur les références (1/2)

```
void main {
  int i = 7;
  int& r = i; // r est une référence vers i.
  r = 9;      // i prend la valeur 9.
  i = 10;     // i et r prennent la valeur 10.
  int j = 0;
  r = j;      // On ne peut pas réassigner les références. r est toujours une référence sur i.
              // r et i valent 0, car &r et &i sont les mêmes.
}
```

Quelques remarques sur les références (2/2)

```
void g(int a, int &r, const int& cr) {
  ++a;
  ++r;
  int x = cr;
}

int main() {
  int x = 0;
  int y = 0;
  int z = 0;

  g(x, y, z); // Au retour, x vaut 0, y vaut 1, z vaut 0
  g(1,2,3) ;  // **Erreur** : une référence a besoin d'une variable qu'elle référence
              //              Or, le paramètre "2" pose souci.
  g(1,y,3);   // OK, y compris pour le paramètre "3" grâce au const-ref
              // Au retour, x vaut 0, y vaut 2, z vaut 0
}
```

Vecteurs

1. Avant-propos : Notion de référence en C++
2. → Vecteurs
3. Erase, tri et lambda expressions

Vecteurs : Déclaration et initialisation

```cpp
std::vector<int> v1(5);          // définit un vecteur de 5 entiers (initialisés à 0)
std::vector<int> v2(5,42);       // définit un vecteur de 5 entiers (initialisés à 42)
std::vector<int> v3{0,1,2,3,4}; // définit un vecteur de 5 entiers allant de 0 à 4.

std::vector<int> v4(5);              // définit un vecteur de 5 entiers (initialisés à 0)
std::iota(v4.begin(), v4.end(), 0); // Initialise ce vecteur de manière séquentielle croissante à partir de 0. Donc, v4 == v3
```

Vecteurs : Quelques fonctions utiles

- Quelques fonctions utiles de la classe vector (cliquez sur la méthode pour accéder à sa *cppreference*):
  - `v[i]` : Accès au $i^{ème}$ élément d'un vecteur (NB : **Aucun *bound checking***).
  - `push_back` : Ajoute à la fin du vecteur. NB : Il existe aussi `emplace_back` pour éviter copie.
  - `pop_back` : Retire de la fin du vecteur.
  - `size` : Retourne le nombre d'éléments du vecteur.
  - `front` : Première valeur dans le vecteur (≡ `v[0]`).
  - `back` : Dernière valeur dans le vecteur (≡ `v[v.size()-1]`).
  - `erase` : Supprime un élément ou un intervalle d'un vecteur, puis bouche les trous avec les éléments suivants (cf. slide "Zoom sur erase").
  - `clear` : Vide un vecteur de tous ses éléments.
  - `assign` : Remplace le contenu d'un vecteur.
  - `insert` : Ajoute des éléments à une certain position d'un vecteur.
- Pour approfondir :
  - Chapitre 16.2 de *Learn C++* et les suivants,
  - Chapitre 19 pages 667-677 du livre.

Parcours

- Boucle for :

```
for (int i = 0; i < v.size(); i++)
  std::cout  << v[i];
```

- Iterator : un itérateur est un objet qui pointe vers des éléments d'un tableau, d'un container (map, vector, ...)

```
for (std::vector<int>::iterator it = v.begin() ; it != v.end(); ++it)
  std::cout << *it;
```

- Range based loop (depuis C++ 11) :

```
for (int i : v)
  std::cout << i;
```

Zoom sur *Range based for loop*

```
for (int i : v)
  std::cout << i;
```

`i` : une variable dont le type est celui des éléments de la séquence représentée par `v`.

```
for (auto i : v)
  std::cout << i;
```

`auto` : Spécifie que le type de la variable sera déduit automatiquement lors de son initialisation à partir du type de `v` (depuis C++ 11).

Attention aux copies cachées dans les boucles

```
//
// **Copie cachée !**
//
for(auto i : v) // Une copie de **chaque** élément est créée : on ne peut pas changer les éléments de v en modifiant i.
  ...

for(auto& i : v) // A utiliser lorsqu'on veut modifier les valeur de v.
  ...

for (const auto& i : v) // Lorsqu'on veut accéder aux éléments de v en lecture seule (pas de copie de chaque élément)
```

NB : Le fonctionnement est le même si le type est spécifié à la place de `auto`. Pour plus de détails, voir cet article de blog.

Erase, tri et lambda expressions

1. Avant-propos : Notion de référence en C++
2. Vecteurs
3. → Erase, tri et lambda expressions

Zoom sur `erase`

```cpp
// Méthode erase appliquée à un vecteur.
// Elle retourne l'itérateur pointant sur l'élément qui suit le dernier élément supprimé.
v.erase (v.begin()+2);           // Supprime un seul élément selon sa position.
v.erase (v.begin(),v.begin()+3); // Suppression des éléments entre deux positions.

// Fonction erase appliquée à un vecteur.
// Elle retourne le nombre d'éléments supprimés.
std::vector<int> v{0, 1, 2, 3, 3, 3, 4, 5, 6, 3, 3, 3, 3};
auto nbErased1 = std::erase(v, 3);
// v contient {0, 1, 2, 4, 5, 6} et nbErased1 contient 7.
auto nbErased2 = std::erase_if(v, [](const auto e) { return e % 2 == 0; });
// v contient {1, 5} et nbErased2 contient 4.
```

- NB :
  - `[](const auto e) { return e % 2 == 0; }` est une lambda expression (cf. slide suivant).
  - `v.erase()` est plus performant que `std::erase()` et `std::erase_if()`.
  - La doc erase est un peu buguée (semble référencer la fonction `remove`).

Tri et lambda expression : Zoom sur tri

- `std::sort(iterator debut, iterator fin, operation de comparaison)`
- ou si tri sur tout le vecteur : `std::ranges::sort(vec, operation de comparaison)`

```cpp
bool cmp_pair(const std::pair<int, int> &e1, const std::pair<int, int> &e2) {
  return e1 < e2;
};

int main() {
  std::vector<std::pair<int, int>> v2{{2, 3}, {0, 1}, {0, 0}};

  std::ranges::sort(v2, cmp_pair); // Besoin de la fonction `cmp_pair`
  // v2 contient {{0, 0}, {0, 1}, {2, 3}}.
  std::ranges::sort(v2, [](const auto &e1, const auto &e2) -> bool { return e1 < e2; }); // Plus besoin de la fonction `cmp_pair`
  // v2 contient {{0, 0}, {0, 1}, {2, 3}}.
  std::ranges::sort(v2, [](const auto &e1, const auto &e2){ return e1 < e2; }); // Plus besoin de la fonction `cmp_pair`, ni de `-> bool`
  // v2 contient {{0, 0}, {0, 1}, {2, 3}}.
  std::ranges::sort(v2, std::less{}); // FYI std::less{} utilisable quand le compilateur peut retrouver l'opérateur `<`
  // v2 contient {{0, 0}, {0, 1}, {2, 3}}.
}
```

Tri et lambda expression : Zoom sur Lambda expression

Lambda : `[capture] (paramètres) -> typeDeRetour {corps}`

- `capture` : Permet de dire les variables extérieures à la lambda qu'on veut pouvoir référencer dans le corps de la lambda :
  - `[]` : Pas d'accès aux variables externes à la lambda expression.
  - `[&]` : Accès à toutes les variables de la fonction par référence.
  - `[=]` : accès à toutes les variables de la fonction par copie.
  - `[a,&b]` : accès à a par copie et à b par référence.
  - NB : Pas de référence-constante disponible (à part en « bidouillant »).
- `paramètres que reçoit la lambda` : Ne pas hésiter à utiliser `const auto &`.
- `-> typeDeRetour` : À indiquer si le compilateur ne peut pas le déduire.
- `corps` : Le corps de votre lambda (ne pas oublier le `;` après chaque instruction).

Questions ?

1. Avant-propos : Notion de référence en C++
2. Vecteurs
3. Erase, tri et lambda expressions

À lire pour la séance prochaine

- **Chapitre 21**